4 – Manipulating Files and Directories

At this point, we are ready for some real work! This chapter will introduce the following commands:

- `cp` – Copy files and directories
- `mv` – Move/rename files and directories
- `mkdir` – Create directories
- `rm` – Remove files and directories
- `ln` – Create hard and symbolic links

These five commands are among the most frequently used Linux commands. They are used for manipulating both files and directories.

Now, to be frank, some of the tasks performed by these commands are more easily done with a graphical file manager. With a file manager, we can drag and drop a file from one directory to another, cut and paste files, delete files, and so on. So why use these old command line programs?

The answer is power and flexibility. While it is easy to perform simple file manipulations with a graphical file manager, complicated tasks can be easier with the command line programs. For example, how could we copy all the HTML files from one directory to another but only copy files that do not exist in the destination directory or are newer than the versions in the destination directory? It's pretty hard with a file manager but pretty easy with the command line.

```
cp -u *.html destination
```

Wildcards

Before we begin using our commands, we need to talk about a shell feature that makes these commands so powerful. Since the shell uses filenames so much, it provides special characters to help us rapidly specify groups of filenames. These special characters are

called *wildcards*. Using wildcards (which is also known as *globbing*) allows us to select filenames based on patterns of characters. Table 4-1 lists the wildcards and what they select.

Table 4-1: Wildcards

Wildcard	Meaning
`*`	Matches any characters
`?`	Matches any single character
`[`*`characters`*`]`	Matches any character that is a member of the set *characters*
`[!`*`characters`*`]`	Matches any character that is not a member of the set *characters*
`[[:`*`class`*`:]]`	Matches any character that is a member of the specified *class*

Table 4-2 lists the most commonly used character classes.

Table 4-2: Commonly Used Character Classes

Character Class	Meaning
`[:alnum:]`	Matches any alphanumeric character
`[:alpha:]`	Matches any alphabetic character
`[:digit:]`	Matches any numeral
`[:lower:]`	Matches any lowercase letter
`[:upper:]`	Matches any uppercase letter

Using wildcards makes it possible to construct sophisticated selection criteria for filenames. Table 4-3 provides some examples of patterns and what they match.

Table 4-3: Wildcard Examples

Pattern	Matches
`*`	All files
`g*`	Any file beginning with “g”
`b*.txt`	Any file beginning with “b” followed by any characters and ending with “.txt”

`Data???`	Any file beginning with “Data” followed by exactly three characters
`[abc]*`	Any file beginning with either an “a”, a “b”, or a “c”
`BACKUP.[0-9][0-9][0-9]`	Any file beginning with “BACKUP.” followed by exactly three numerals
`[[:upper:]]*`	Any file beginning with an uppercase letter
`[![:digit:]]*`	Any file not beginning with a numeral
`*[[:lower:]123]`	Any file ending with a lowercase letter or the numerals “1”, “2”, or “3”

Wildcards can be used with any command that accepts filenames as arguments, but we’ll talk more about that in Chapter 7, "Seeing the World As the Shell Sees It.

Character Ranges

If you are coming from another Unix-like environment or have been reading some other books on this subject, you may have encountered the `[A-Z]` and `[a-z]` character range notations. These are traditional Unix notations and worked in older versions of Linux as well. They can still work, but you have to be careful with them because they will not produce the expected results unless properly configured. For now, you should avoid using them and use character classes instead.

Wildcards Work in the GUI Too

Wildcards are especially valuable not only because they are used so frequently on the command line, but because they are also supported by some graphical file managers.

- In Nautilus (the file manager for GNOME), you can select files using the Edit/Select Pattern menu item. Just enter a file selection pattern with wildcards and the files in the currently viewed directory will be highlighted for selection.

- In some versions of Dolphin and Konqueror (the file managers for KDE), you can enter wildcards directly on the location bar. For example, if you want to see all the files starting with a lowercase “u” in the /usr/bin directory, enter “/usr/bin/u*” in the location bar and it will display the result.

Many ideas originally found in the command line interface make their way into the graphical interface, too. It is one of the many things that make the Linux desktop so powerful.

`mkdir` – Create Directories

The `mkdir` command is used to create directories. It works like this:

```
mkdir directory...
```

A note on notation: When three periods follow an argument in the description of a command (as above), it means that the argument can be repeated, thus the following command:

```
mkdir dir1
```

would create a single directory named `dir1`, while the following:

```
mkdir dir1 dir2 dir3
```

would create three directories named `dir1`, `dir2`, and `dir3`.

`cp` – Copy Files and Directories

The `cp` command copies files or directories. It can be used two different ways. The following:

```
cp item1 item2
```

copies the single file or directory `item1` to the file or directory `item2` and the following:

```
cp item... directory
```

copies multiple items (either files or directories) into a directory.

Useful Options and Examples

Table 4-4 lists some of the commonly used options for `cp`.

Table 4-4: `cp` Options

Option	Long Option	Meaning
`-a`	`--archive`	Copy the files and directories and all of their attributes, including ownerships and permissions. Normally, copies take on the default attributes of the user performing the copy. We'll take a look at file permissions in Chapter 9 "Permissions."
`-i`	`--interactive`	Before overwriting an existing file, prompt the user for confirmation. **If this option is not specified, `cp` will silently (meaning there will be no warning) overwrite files.**
`-r`	`--recursive`	Recursively copy directories and their contents. This option (or the `-a` option) is required when copying directories.
`-u`	`--update`	When copying files from one directory to another, only copy files that either don't exist or are newer than the existing corresponding files, in the destination directory. This is useful when copying large numbers of files as it skips files that don't need to be copied.
`-v`	`--verbose`	Display informative messages as the copy is performed.

Table 4-5: `cp` Examples

Command	Results
`cp file1 file2`	Copy *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it

	is created.
`cp -i file1 file2`	Same as previous command, except that if *file2* exists, the user is prompted before it is overwritten.
`cp file1 file2 dir1`	Copy *file1* and *file2* into directory *dir1*. The directory *dir1* must already exist.
`cp dir1/* dir2`	Using a wildcard, copy all the files in *dir1* into *dir2*. The directory *dir2* must already exist.
`cp -r dir1 dir2`	Copy the contents of directory *dir1* to directory *dir2*. If directory *dir2* does not exist, it is created and, after the copy, will contain the same contents as directory *dir1*. If directory *dir2* does exist, then directory *dir1* (and its contents) will be copied into *dir2*.

`mv` – Move and Rename Files

The `mv` command performs both file moving and file renaming, depending on how it is used. In either case, the original filename no longer exists after the operation. `mv` is used in much the same way as `cp`, as shown here:

```
mv item1 item2
```

to move or rename the file or directory `item1` to `item2` or:

```
mv item... directory
```

to move one or more items from one directory to another.

Useful Options and Examples

`mv` shares many of the same options as `cp` as described in Table 4-6.

Table 4-6: `mv` Options

Option	Long Option	Meaning
`-i`	`--interactive`	Before overwriting an existing file, prompt the

		user for confirmation. **If this option is not specified, `mv` will silently overwrite files.**
`-u`	`--update`	When moving files from one directory to another, only move files that either don't exist, or are newer than the existing corresponding files in the destination directory.
`-v`	`--verbose`	Display informative messages as the move is performed.

Table 4-7 provides some examples of `mv` usage.

Table 4-7: `mv` Examples

Command	Results
`mv` *`file1 file2`*	Move *file1* to *file2*. **If *file2* exists, it is overwritten with the contents of *file1*.** If *file2* does not exist, it is created. **In either case, *file1* ceases to exist.**
`mv -i` *`file1 file2`*	Same as the previous command, except that if *file2* exists, the user is prompted before it is overwritten.
`mv` *`file1 file2 dir1`*	Move *file1* and *file2* into directory *dir1*. The dierctory *dir1* must already exist.
`mv` *`dir1 dir2`*	If directory *dir2* does not exist, create directory *dir2* and move the contents of directory *dir1* into *dir2* and delete directory *dir1*. If directory *dir2* does exist, move directory *dir1* (and its contents) into directory *dir2*.

`rm` – Remove Files and Directories

The `rm` command is used to remove (delete) files and directories, as shown here:

```
rm item...
```

where `item` is one or more files or directories.

Useful Options and Examples

Table 4-8 describes some of the common options for `rm`.

Table 4-8: `rm` Options

Option	Long Option	Meaning
`-i`	`--interactive`	Before deleting an existing file, prompt the user for confirmation. **If this option is not specified, `rm` will silently delete files.**
`-r`	`--recursive`	Recursively delete directories. This means that if a directory being deleted has subdirectories, delete them too. To delete a directory, this option must be specified.
`-f`	`--force`	Ignore nonexistent files and do not prompt. This overrides the `--interactive` option.
`-v`	`--verbose`	Display informative messages as the deletion is performed.

Table 4-9 provides some examples of using the `rm` command.

Table 4-9: `rm` Examples

Command	Results
`rm` *`file1`*	Delete *file1* silently.
`rm -i` *`file1`*	Same as the previous command, except that the user is prompted for confirmation before the deletion is performed.
`rm -r` *`file1 dir1`*	Delete *file1* and *dir1* and its contents.
`rm -rf` *`file1 dir1`*	Same as the previous command, except that if either *file1* or *dir1* do not exist, `rm` will continue silently.

Be Careful with `rm`!

Unix-like operating systems such as Linux do not have an undelete command. Once you delete something with `rm`, it's gone. Linux assumes you're smart and you know what you're doing.

Be particularly careful with wildcards. Consider this classic example. Let's say you want to delete just the HTML files in a directory. To do this, you type the following:

```
rm *.html
```

This is correct, but if you accidentally place a space between the `*` and the `.html` like so:

```
rm * .html
```

the `rm` command will delete all the files in the directory and then complain that there is no file called `.html`.

Here is a useful tip: whenever you use wildcards with `rm` (besides carefully checking your typing!), test the wildcard first with `ls`. This will let you see the files that will be deleted. Then press the up arrow key to recall the command and replace `ls` with `rm`.

`ln` – Create Links

The `ln` command is used to create either hard or symbolic links. It is used in one of two ways. The following creates a hard link:

```
ln file link
```

The following creates a symbolic link:

```
ln -s item link
```

to create a symbolic link where `item` is either a file or a directory.

Hard Links

Hard links are the original Unix way of creating links, compared to symbolic links, which

are more modern. By default, every file has a single hard link that gives the file its name. When we create a hard link, we create an additional directory entry for a file. Hard links have two important limitations:

1. A hard link cannot reference a file outside its own file system. This means a link cannot reference a file that is not on the same disk partition as the link itself.
2. A hard link may not reference a directory.

A hard link is indistinguishable from the file itself. Unlike a symbolic link, when we list a directory containing a hard link we will see no special indication of the link. When a hard link is deleted, the link is removed but the contents of the file itself continue to exist (that is, its space is not deallocated) until all links to the file are deleted.

It is important to be aware of hard links because you might encounter them from time to time, but modern practice prefers symbolic links, which we will cover next.

Symbolic Links

Symbolic links were created to overcome the limitations of hard links. Symbolic links work by creating a special type of file that contains a text pointer to the referenced file or directory. In this regard, they operate in much the same way as a Windows shortcut, though of course they predate the Windows feature by many years.

A file pointed to by a symbolic link, and the symbolic link itself are largely indistinguishable from one another. For example, if we write something to the symbolic link, the referenced file is written to. However when we delete a symbolic link, only the link is deleted, not the file itself. If the file is deleted before the symbolic link, the link will continue to exist but will point to nothing. In this case, the link is said to be *broken*. In many implementations, the `ls` command will display broken links in a distinguishing color, such as red, to reveal their presence.

The concept of links can seem confusing, but hang in there. We're going to try all this stuff and it will, hopefully, become clear.

Let's Build a Playground

Since we are going to do some real file manipulation, let's build a safe place to “play” with our file manipulation commands. First we need a directory to work in. We'll create one in our home directory and call it `playground`.

Creating Directories

The `mkdir` command is used to create a directory. To create our playground directory we will first make sure we are in our home directory and will then create the new directory.

```
[me@linuxbox ~]$ cd
[me@linuxbox ~]$ mkdir playground
```

To make our playground a little more interesting, let's create a couple of directories inside it called `dir1` and `dir2`. To do this, we will change our current working directory to `playground` and execute another `mkdir`.

```
[me@linuxbox ~]$ cd playground
[me@linuxbox playground]$ mkdir dir1 dir2
```

Notice that the `mkdir` command will accept multiple arguments allowing us to create both directories with a single command.

Copying Files

Next, let's get some data into our playground. We'll do this by copying a file. Using the `cp` command, we'll copy the `passwd` file from the `/etc` directory to the current working directory.

```
[me@linuxbox playground]$ cp /etc/passwd .
```

Notice how we used shorthand for the current working directory, the single trailing period. So now if we perform an `ls`, we will see our file.

```
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me  me 4096 2018-01-10 16:40 dir1
drwxrwxr-x 2 me  me 4096 2018-01-10 16:40 dir2
-rw-r--r-- 1 me  me 1650 2018-01-10 16:07 passwd
```

Now, just for fun, let's repeat the copy using the “-v” option (verbose) to see what it does.

```
[me@linuxbox playground]$ cp -v /etc/passwd .
`/etc/passwd' -> `./passwd'
```

The `cp` command performed the copy again, but this time displayed a concise message indicating what operation it was performing. Notice that `cp` overwrote the first copy

without any warning. Again this is a case of `cp` assuming that we know what we're doing. To get a warning, we'll include the “-i” (interactive) option.

```
[me@linuxbox playground]$ cp -i /etc/passwd .
cp: overwrite `./passwd'?
```

Responding to the prompt by entering a `y` will cause the file to be overwritten, any other character (for example, `n`) will cause `cp` to leave the file alone.

Moving and Renaming Files

Now, the name `passwd` doesn't seem very playful and this is a playground, so let's change it to something else.

```
[me@linuxbox playground]$ mv passwd fun
```

Let's pass the fun around a little by moving our renamed file to each of the directories and back again. The following moves it first to the directory `dir1`:

```
[me@linuxbox playground]$ mv fun dir1
```

The following then moves it from `dir1` to `dir2`:

```
[me@linuxbox playground]$ mv dir1/fun dir2
```

Finally, the following brings it back to the current working directory:

```
[me@linuxbox playground]$ mv dir2/fun .
```

Next, let's see the effect of `mv` on directories. First we will move our data file into `dir1` again, like this:

```
[me@linuxbox playground]$ mv fun dir1
```

Then we move `dir1` into `dir2` and confirm it with `ls.`

```
[me@linuxbox playground]$ mv dir1 dir2
[me@linuxbox playground]$ ls -l dir2
total 4
drwxrwxr-x 2 me  me   4096 2018-01-11 06:06 dir1
[me@linuxbox playground]$ ls -l dir2/dir1
total 4
-rw-r--r-- 1 me  me   1650 2018-01-10 16:33 fun
```

Note that since `dir2` already existed, `mv` moved `dir1` into `dir2`. If `dir2` had not existed, `mv` would have renamed `dir1` to `dir2`. Lastly, let's put everything back.

```
[me@linuxbox playground]$ mv dir2/dir1 .
[me@linuxbox playground]$ mv dir1/fun .
```

Creating Hard Links

Now we'll try some links. We'll first create some hard links to our data file like so:

```
[me@linuxbox playground]$ ln fun fun-hard
[me@linuxbox playground]$ ln fun dir1/fun-hard
[me@linuxbox playground]$ ln fun dir2/fun-hard
```

So now we have four instances of the file `fun`. Let's take a look at our playground directory.

```
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2018-01-14 16:17 dir1
drwxrwxr-x 2 me   me   4096 2018-01-14 16:17 dir2
-rw-r--r-- 4 me   me   1650 2018-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2018-01-10 16:33 fun-hard
```

One thing we notice is that both the second fields in the listings for `fun` and `fun-hard` contain a `4` which is the number of hard links that now exist for the file. Remember that a file will always have at least one link because the file's name is created by a link. So, how do we know that `fun` and `fun-hard` are, in fact, the same file? In this case, `ls` is not very helpful. While we can see that `fun` and `fun-hard` are both the same size (field 5), our listing provides no way to be sure. To solve this problem, we're going to have to dig a

little deeper.

When thinking about hard links, it is helpful to imagine that files are made up of two parts.

1. The data part containing the file's contents.
2. The name part that holds the file's name.

When we create hard links, we are actually creating additional name parts that all refer to the same data part. The system assigns a chain of disk blocks to what is called an *inode*, which is then associated with the name part. Each hard link therefore refers to a specific inode containing the file's contents.

The `ls` command has a way to reveal this information. It is invoked with the `-i` option.

```
[me@linuxbox playground]$ ls -li
total 16
12353539 drwxrwxr-x 2 me   me   4096 2018-01-14 16:17 dir1
12353540 drwxrwxr-x 2 me   me   4096 2018-01-14 16:17 dir2
12353538 -rw-r--r-- 4 me   me   1650 2018-01-10 16:33 fun
12353538 -rw-r--r-- 4 me   me   1650 2018-01-10 16:33 fun-hard
```

In this version of the listing, the first field is the inode number and, as we can see, both `fun` and `fun-hard` share the same inode number, which confirms they are the same file.

Creating Symbolic Links

Symbolic links were created to overcome the two disadvantages of hard links.

1. Hard links cannot span physical devices.
2. Hard links cannot reference directories, only files.

Symbolic links are a special type of file that contains a text pointer to the target file or directory.

Creating symbolic links is similar to creating hard links.

```
[me@linuxbox playground]$ ln -s fun fun-sym
[me@linuxbox playground]$ ln -s ../fun dir1/fun-sym
[me@linuxbox playground]$ ln -s ../fun dir2/fun-sym
```

The first example is pretty straightforward; we simply add the “-s” option to create a

symbolic link rather than a hard link. But what about the next two? Remember, when we create a symbolic link, we are creating a text description of where the target file is relative to the symbolic link. It's easier to see if we look at the `ls` output shown here:

```
[me@linuxbox playground]$ ls -l dir1
total 4
-rw-r--r-- 4 me   me   1650 2018-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      6 2018-01-15 15:17 fun-sym -> ../fun
```

The listing for `fun-sym` in `dir1` shows that it is a symbolic link by the leading `l` in the first field and that it points to `../fun`, which is correct. Relative to the location of `fun-sym`, `fun` is in the directory above it. Notice too, that the length of the symbolic link file is 6, the number of characters in the string `../fun` rather than the length of the file to which it is pointing.

When creating symbolic links, we can either use absolute pathnames, as shown here:

```
[me@linuxbox playground]$ ln -s /home/me/playground/fun dir1/fun-sym
```

or relative pathnames, as we did in our earlier example. In most cases, using relative pathnames is more desirable because it allows a directory tree containing symbolic links and their referenced files to be renamed and/or moved without breaking the links.

In addition to regular files, symbolic links can also reference directories.

```
[me@linuxbox playground]$ ln -s dir1 dir1-sym
[me@linuxbox playground]$ ls -l
total 16
drwxrwxr-x 2 me   me   4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2018-01-15 15:17 dir2
-rw-r--r-- 4 me   me   1650 2018-01-10 16:33 fun
-rw-r--r-- 4 me   me   1650 2018-01-10 16:33 fun-hard
lrwxrwxrwx 1 me   me      3 2018-01-15 15:15 fun-sym -> fun
```

Removing Files and Directories

As we covered earlier, the `rm` command is used to delete files and directories. We are going to use it to clean up our playground a little bit. First, let's delete one of our hard links.

```
[me@linuxbox playground]$ rm fun-hard
[me@linuxbox playground]$ ls -l
total 12
drwxrwxr-x 2 me   me   4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2018-01-15 15:17 dir2
-rw-r--r-- 3 me   me   1650 2018-01-10 16:33 fun
lrwxrwxrwx 1 me   me      3 2018-01-15 15:15 fun-sym -> fun
```

That worked as expected. The file `fun-hard` is gone and the link count shown for `fun` is reduced from four to three, as indicated in the second field of the directory listing. Next, we'll delete the file `fun`, and just for enjoyment, we'll include the `-i` option to show what that does.

```
[me@linuxbox playground]$ rm -i fun
rm: remove regular file `fun'?
```

Enter `y` at the prompt and the file is deleted. But let's look at the output of `ls` now. Notice what happened to `fun-sym`? Since it's a symbolic link pointing to a now-nonexistent file, the link is *broken*.

```
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me   me   4096 2018-01-15 15:17 dir1
lrwxrwxrwx 1 me   me      4 2018-01-16 14:45 dir1-sym -> dir1
drwxrwxr-x 2 me   me   4096 2018-01-15 15:17 dir2
lrwxrwxrwx 1 me   me      3 2018-01-15 15:15 fun-sym -> fun
```

Most Linux distributions configure `ls` to display broken links. The presence of a broken link is not in and of itself dangerous, but it is rather messy. If we try to use a broken link we will see this:

```
[me@linuxbox playground]$ less fun-sym
fun-sym: No such file or directory
```

Let's clean up a little. We'll delete the symbolic links here:

```
[me@linuxbox playground]$ rm fun-sym dir1-sym
[me@linuxbox playground]$ ls -l
total 8
drwxrwxr-x 2 me   me   4096 2018-01-15 15:17 dir1
drwxrwxr-x 2 me   me   4096 2018-01-15 15:17 dir2
```

One thing to remember about symbolic links is that most file operations are carried out on the link's target, not the link itself. `rm` is an exception. When we delete a link, it is the link that is deleted, not the target.

Finally, we will remove our playground. To do this, we will return to our home directory and use `rm` with the recursive option (`-r`) to delete `playground` and all of its contents, including its subdirectories.

```
[me@linuxbox playground]$ cd
[me@linuxbox ~]$ rm -r playground
```

Creating Symlinks With The GUI

The file managers in both GNOME and KDE provide an easy and automatic method of creating symbolic links. With GNOME, holding the Ctrl+Shift keys while dragging a file will create a link rather than copying (or moving) the file. In KDE, a small menu appears whenever a file is dropped, offering a choice of copying, moving, or linking the file.

Summing Up

We've covered a lot of ground here and it will take a while for it all to fully sink in. Perform the playground exercise over and over until it makes sense. It is important to get a good understanding of basic file manipulation commands and wildcards. Feel free to expand on the playground exercise by adding more files and directories, using wildcards to specify files for various operations. The concept of links is a little confusing at first, but take the time to learn how they work. They can be a real lifesaver.

Further Reading

- A discussion of symbolic links: http://en.wikipedia.org/wiki/Symbolic_link

5 – Working with Commands

Up to this point, we have seen a series of mysterious commands, each with its own mysterious options and arguments. In this chapter, we will attempt to remove some of that mystery and even create our own commands. The commands introduced in this chapter are:

- `type` – Indicate how a command name is interpreted
- `which` – Display which executable program will be executed
- `help` – Get help for shell builtins
- `man` – Display a command's manual page
- `apropos` – Display a list of appropriate commands
- `info` – Display a command's info entry
- `whatis` – Display one-line manual page descriptions
- `alias` – Create an alias for a command

What Exactly Are Commands?

A command can be one of four different things:

1. **An executable program** like all those files we saw in `/usr/bin`. Within this category, programs can be *compiled binaries* such as programs written in C and C++, or programs written in *scripting languages* such as the shell, Perl, Python, Ruby, and so on.
2. **A command built into the shell itself**. `bash` supports a number of commands internally called *shell builtins*. The `cd` command, for example, is a shell builtin.
3. **A shell function.** Shell functions are miniature shell scripts incorporated into the *environment*. We will cover configuring the environment and writing shell functions in later chapters, but for now, just be aware that they exist.
4. **An alias.** Aliases are commands that we can define ourselves, built from other commands.

Identifying Commands

It is often useful to know exactly which of the four kinds of commands is being used and Linux provides a couple of ways to find out.

`type` – Display a Command's Type

The `type` command is a shell builtin that displays the kind of command the shell will execute, given a particular command name. It works like this:

```
type command
```

where “command” is the name of the command we want to examine. Here are some examples:

```
[me@linuxbox ~]$ type type
type is a shell builtin
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
[me@linuxbox ~]$ type cp
cp is /bin/cp
```

Here we see the results for three different commands. Notice the one for `ls` (taken from a Fedora system) and how the `ls` command is actually an alias for the `ls` command with the “--color=tty” option added. Now we know why the output from `ls` is displayed in color!

`which` – Display an Executable's Location

Sometimes there is more than one version of an executable program installed on a system. While this is not common on desktop systems, it's not unusual on large servers. To determine the exact location of a given executable, the `which` command is used.

```
[me@linuxbox ~]$ which ls
/bin/ls
```

`which` only works for executable programs, not builtins nor aliases that are substitutes for actual executable programs. When we try to use `which` on a shell builtin for example, `cd`, we either get no response or get an error message:

```
[me@linuxbox ~]$ which cd
/usr/bin/which: no cd in (/usr/local/bin:/usr/bin:/bin:/usr/local
/games:/usr/games)
```

This response is a fancy way of saying “command not found.”

Getting a Command's Documentation

With this knowledge of what a command is, we can now search for the documentation available for each kind of command.

`help` – Get Help for Shell Builtins

`bash` has a built-in help facility available for each of the shell builtins. To use it, type “help” followed by the name of the shell builtin. Here is an example:

```
[me@linuxbox ~]$ help cd
cd: cd [-L|[-P [-e]] [-@]] [dir]
    Change the shell working directory.

    Change the current directory to DIR.  The default DIR is the
    value of the HOME shell variable.

    The variable CDPATH defines the search path for the directory
    containing DIR.  Alternative directory names in CDPATH are
    separated by a colon (:). A null directory name is the same as
    the current directory.  If DIR begins with a slash (/), then
    CDPATH is not used.

    If the directory is not found, and the shell option `cdable_vars'
    is set, the word is assumed to be  a variable name.  If that
    variable has a value, its value is used for DIR.

    Options:
       -L    force symbolic links to be followed: resolve symbolic
             links in DIR after processing instances of `..'
       -P    use the physical directory structure without following
             symbolic links: resolve symbolic links in DIR before
             processing instances of `..'
       -e    if the -P option is supplied, and the current working
             directory cannot be determined successfully, exit with
             a non-zero status
```

```
       -@    on systems that support it, present a file with extended
             attributes as a directory containing the file attributes


     The default is to follow symbolic links, as if `-L' were
    specified. `..' is processed by removing the immediately previous
    pathname component back to a slash or the beginning of DIR.

    Exit Status:
    Returns 0 if the directory is changed, and if $PWD is set
    successfully when -P is used; non-zero otherwise.
```

A note on notation: When square brackets appear in the description of a command's syntax, they indicate optional items. A vertical bar character indicates mutually exclusive items. In the case of the `cd` command above:

```
cd [-L|[-P[-e]]] [dir]
```

This notation says that the command `cd` may be followed optionally by either a “-L” or a “-P” and further, if the “-P” option is specified the “-e” option may also be included followed by the optional argument “dir”.

While the output of `help` for the `cd` commands is concise and accurate, it is by no means tutorial and as we can see, it also seems to mention a lot of things we haven't talked about yet! Don't worry. We'll get there.

`--help` – Display Usage Information

Many executable programs support a “--help” option that displays a description of the command's supported syntax and options. For example:

```
[me@linuxbox ~]$ mkdir --help
Usage: mkdir [OPTION] DIRECTORY...
Create the DIRECTORY(ies), if they do not already exist.

  -Z, --context=CONTEXT (SELinux) set security context to CONTEXT
Mandatory arguments to long options are mandatory for short options
too.
  -m, --mode=MODE   set file mode (as in chmod), not a=rwx – umask
  -p, --parents     no error if existing, make parent directories as
                    needed
  -v, --verbose     print a message for each created directory
      --help        display this help and exit
```

```
      --version     output version information and exit
Report bugs to <bug-coreutils@gnu.org>.
```

Some programs don't support the “--help” option, but try it anyway. Often it results in an error message that will reveal the same usage information.

man – Display a Program's Manual Page

Most executable programs intended for command line use provide a formal piece of documentation called a *manual* or *man page*. A special paging program called `man` is used to view them. It is used like this:

```
man program
```

where “program” is the name of the command to view.

Man pages vary somewhat in format but generally contain the following:

- A title (the page’s name)
- A synopsis of the command's syntax
- A description of the command's purpose
- A listing and description of each of the command's options

Man pages, however, do not usually include examples, and are intended as a reference, not a tutorial. As an example, let's try viewing the man page for the `ls` command:

```
[me@linuxbox ~]$ man ls
```

On most Linux systems, `man` uses `less` to display the manual page, so all of the familiar `less` commands work while displaying the page.

The “manual” that `man` displays is broken into sections and covers not only user commands but also system administration commands, programming interfaces, file formats and more. Table 5-1 describes the layout of the manual.

Table 5-1: Man Page Organization

Section	Contents
1	User commands

2	Programming interfaces for kernel system calls
3	Programming interfaces to the C library
4	Special files such as device nodes and drivers
5	File formats
6	Games and amusements such as screen savers
7	Miscellaneous
8	System administration commands

Sometimes we need to refer to a specific section of the manual to find what we are looking for. This is particularly true if we are looking for a file format that is also the name of a command. Without specifying a section number, we will always get the first instance of a match, probably in section 1. To specify a section number, we use `man` like this:

```
man section search_term
```

Here's an example:

```
[me@linuxbox ~]$ man 5 passwd
```

This will display the man page describing the file format of the `/etc/passwd` file.

`apropos` – Display Appropriate Commands

It is also possible to search the list of man pages for possible matches based on a search term. It's crude but sometimes helpful. Here is an example of a search for man pages using the search term *partition*:

```
[me@linuxbox ~]$ apropos partiton
addpart (8)          - simple wrapper around the "add partition"...
all-swaps (7)        - event signalling that all swap partitions...
cfdisk (8)           - display or manipulate disk partition table
cgdisk (8)           - Curses-based GUID partition table (GPT)...
delpart (8)          - simple wrapper around the "del partition"...
fdisk (8)            - manipulate disk partition table
fixparts (8)         - MBR partition table repair utility
```

```
gdisk (8)            - Interactive GUID partition table (GPT)...
mpartition (1)       - partition an MSDOS hard disk
partprobe (8)        - inform the OS of partition table changes
partx (8)            - tell the Linux kernel about the presence...
resizepart (8)       - simple wrapper around the "resize partition...

sfdisk (8)           - partition table manipulator for Linux
sgdisk (8)           - Command-line GUID partition table (GPT)...
```

The first field in each line of output is the name of the man page, and the second field shows the section. Note that the `man` command with the "-k" option performs the same function as `apropos`.

`whatis` – Display One-line Manual Page Descriptions

The `whatis` program displays the name and a one-line description of a man page matching a specified keyword:

```
[me@linuxbox ~]$ whatis ls
ls                   (1)  - list directory contents
```

The Most Brutal Man Page Of Them All

As we have seen, the manual pages supplied with Linux and other Unix-like systems are intended as reference documentation and not as tutorials. Many man pages are hard to read, but I think that the grand prize for difficulty has got to go to the man page for `bash`. As I was doing research for this book, I gave the `bash` man page careful review to ensure that I was covering most of its topics. When printed, it's more than 80 pages long and extremely dense, and its structure makes absolutely no sense to a new user.

On the other hand, it is very accurate and concise, as well as being extremely complete. So check it out if you dare and look forward to the day when you can read it and it all makes sense.

`info` – Display a Program's Info Entry

The GNU Project provides an alternative to man pages for their programs, called "info."

Info manuals are displayed with a reader program named, appropriately enough, `info`. Info pages are *hyperlinked* much like web pages. Here is a sample:

```
File: coreutils.info,  Node: ls invocation,  Next: dir invocation,
Up: Directory listing
10.1 `ls': List directory contents
==================================
The `ls' program lists information about files (of any type,
including directories).  Options and file arguments can be intermixed
arbitrarily, as usual.
   For non-option command-line arguments that are directories, by
default `ls' lists the contents of directories, not recursively, and
omitting files with names beginning with `.'.  For other non-option
arguments, by default `ls' lists just the filename.  If no non-option
argument is specified, `ls' operates on the current directory, acting
as if it had been invoked with a single argument of `.'.
   By default, the output is sorted alphabetically, according to the
--zz-Info: (coreutils.info.gz)ls invocation, 63 lines --Top---------
```

The `info` program reads *info files*, which are tree structured into individual *nodes*, each containing a single topic. Info files contain hyperlinks that can move the reader from node to node. A hyperlink can be identified by its leading asterisk and is activated by placing the cursor upon it and pressing the Enter key.

To invoke `info`, type `info` followed optionally by the name of a program. Table 5-2 describes the commands used to control the reader while displaying an info page.

Table 5-2: `info` Commands

Command	Action
`?`	Display command help
`PgUp` or `Backspace`	Display previous page
`PgDn` or `Space`	Display next page
`n`	Next - Display the next node
`p`	Previous - Display the previous node
`u`	Up - Display the parent node of the currently displayed node, usually a menu
`Enter`	Follow the hyperlink at the cursor location

q	Quit

Most of the command line programs we have discussed so far are part of the GNU Project's *coreutils* package, so typing the following:

```
[me@linuxbox ~]$ info coreutils
```

will display a menu page with hyperlinks to each program contained in the coreutils package.

README and Other Program Documentation Files

Many software packages installed on our system have documentation files residing in the `/usr/share/doc` directory. Most of these are stored in plain text format and can be viewed with `less`. Some of the files are in HTML format and can be viewed with a web browser. We may encounter some files ending with a ".gz" extension. This indicates that they have been compressed with the `gzip` compression program. The gzip package includes a special version of `less` called `zless` that will display the contents of gzip-compressed text files.

Creating Our Own Commands with `alias`

Now for our first experience with programming! We will create a command of our own using the `alias` command. But before we start, we need to reveal a small command line trick. It's possible to put more than one command on a line by separating each command with a semicolon. It works like this:

```
command1; command2; command3...
```

Here's the example we will use:

```
[me@linuxbox ~]$ cd /usr; ls; cd -
bin  games  include  lib  local  sbin  share  src
/home/me
[me@linuxbox ~]$
```

As we can see, we have combined three commands on one line. First we change directory to `/usr` then list the directory and finally return to the original directory (by using `'cd`

`-'`) so we end up where we started. Now let's turn this sequence into a new command using `alias`. The first thing we have to do is dream up a name for our new command. Let's try “test”. Before we do that, it would be a good idea to find out if the name “test” is already being used. To find out, we can use the `type` command again:

```
[me@linuxbox ~]$ type test
test is a shell builtin
```

Oops! The name `test` is already taken. Let's try `foo`:

```
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

Great! “foo” is not taken. So let's create our alias:

```
[me@linuxbox ~]$ alias foo='cd /usr; ls; cd -'
```

Notice the structure of this command shown here:

```
alias name='string'
```

After the command `alias`, we give alias a name followed immediately (no whitespace allowed) by an equal sign, followed immediately by a quoted string containing the meaning to be assigned to the name. After we define our alias, we can use it anywhere the shell would expect a command. Let's try it:

```
[me@linuxbox ~]$ foo
bin  games  include  lib  local  sbin  share  src
/home/me
[me@linuxbox ~]$
```

We can also use the `type` command again to see our alias:

```
[me@linuxbox ~]$ type foo
foo is aliased to `cd /usr; ls; cd -'
```

To remove an alias, the `unalias` command is used, like so:

```
[me@linuxbox ~]$ unalias foo
[me@linuxbox ~]$ type foo
bash: type: foo: not found
```

While we purposefully avoided naming our alias with an existing command name, it is not uncommon to do so. This is often done to apply a commonly desired option to each invocation of a common command. For instance, we saw earlier how the `ls` command is often aliased to add color support:

```
[me@linuxbox ~]$ type ls
ls is aliased to `ls --color=tty'
```

To see all the aliases defined in the environment, use the `alias` command without arguments. Here are some of the aliases defined by default on a Fedora system. Try to figure out what they all do:

```
[me@linuxbox ~]$ alias
alias l.='ls -d .* --color=tty'
alias ll='ls -l --color=tty'
alias ls='ls --color=tty'
```

There is one tiny problem with defining aliases on the command line. They vanish when our shell session ends. In Chapter 11, "The Environment", we will see how to add our own aliases to the files that establish the environment each time we log on, but for now, enjoy the fact that we have taken our first, albeit tiny, step into the world of shell programming!

Summing Up

Now that we have learned how to find the documentation for commands, go and look up the documentation for all the commands we have encountered so far. Study what additional options are available and try them!

Further Reading

There are many online sources of documentation for Linux and the command line. Here are some of the best:

- The *Bash Reference Manual* is a reference guide to the `bash` shell. It's still a reference work but contains examples and is easier to read than the `bash` man page.
  http://www.gnu.org/software/bash/manual/bashref.html
- The *Bash FAQ* contains answers to frequently asked questions regarding `bash`. This list is aimed at intermediate to advanced users, but contains a lot of good information.
  http://mywiki.wooledge.org/BashFAQ
- The GNU Project provides extensive documentation for its programs, which form the core of the Linux command line experience. You can see a complete list here:
  http://www.gnu.org/manual/manual.html
- Wikipedia has an interesting article on man pages:
  http://en.wikipedia.org/wiki/Man_page